Benchmarking web applications in different architectural variants

JOSÉ FRANCISCO DOMINGOS REIS CUNHA
(Licenciado)

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientador: Prof. Doutor Miguel Gamboa de Carvalho

Júri:

Presidente:	Prof. Doutor Nuno Miguel Soares Datia
Vogais:	Prof. Doutor Filipe Bastos de Freitas
	Prof. Doutor Miguel Gamboa de Carvalho

Outubro 2024

Benchmarking web applications in different architectural variants

JOSÉ FRANCISCO DOMINGOS REIS CUNHA

(Licenciado)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientador: Prof. Doutor Miguel Gamboa de Carvalho, ISEL

Júri:

Presidente:	Prof. Doutor Nuno Miguel Soares Datia, ISEL
Vogais:	Prof. Doutor Filipe Bastos de Freitas, ISEL
	Prof. Doutor Miguel Gamboa de Carvalho, ISEL

Outubro 2024

Acknowledgements

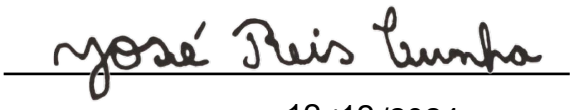
I would like to express my gratitude to Instituto Superior de Engenharia de Lisboa for the opportunity to pursue this research. Special thanks to my advisor, Dr. Miguel Gamboa de Carvalho, for their guidance and support throughout this journey.

I would like to extend my heartfelt thanks to my family and friends for their unwavering support and encouragement during this journey. A special mention goes to my wife, Jessica Cunha, who have been by my side through thick and thin.

Statement of integrity

I declare that this **dissertation** is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author



Lisbon, 12 / 12 /2024

Benchmarking web applications in different architectural variants

This document was created using the (pdf)LaTeX processor, based in the "iselthesis" template [5], developed at the DEETC of ISEL-IPL.

Abstract

This dissertation addresses the challenge of achieving high performance in web applications that handle a significant number of simultaneous users while maintaining acceptable response times. The approach involves the implementation of a configurable testbed within the Java ecosystem, utilizing a web API and JMeter for performance testing.

The project aims to compare a multi-threaded approach, represented by Spring MVC, with an event-driven approach, exemplified by WebFlux. A reverse proxy will be employed to distribute requests to multiple APIs, implemented in the testbed, facilitating this comparative analysis and enhancing the overall system performance. This investigation will enable conclusions about the performance characteristics of both frameworks, contributing to a deeper understanding of their capabilities in high-load scenarios.

An extensive background on web application performance, scalability techniques, and the intricacies of load testing will support this investigation, leading to conclusion for future developments in this field.

Keywords: Benchmarking; Web frameworks; Non-blocking IO;

Resumo

Esta dissertação aborda o desafio de alcançar alta performance em aplicações web que lidam com um número significativo de utilizadores simultâneos, mantendo tempos de resposta aceitáveis. A abordagem envolve a implementação de um ambiente de teste configurável dentro do ecossistema Java, utilizando uma API web e JMeter para testes de performance.

O projeto tem como objetivo comparar uma abordagem multi-threaded, representada pelo Spring MVC, com uma abordagem event-driven, exemplificada pelo WebFlux. Um reverse proxy será utilizado para distribuir pedidos a várias APIs, implementadas no ambiente de teste, facilitando essa análise comparativa e melhorando a performance geral do sistema. Esta investigação permitirá conclusões sobre as características de desempenho de ambas as frameworks, contribuindo para uma compreensão mais profunda de suas capacidades em cenários de carga.

Um amplo contexto sobre desempenho de aplicações web, técnicas de escalabilidade e as complexidades do teste de carga apoiarão esta investigação, levando a conclusões para desenvolvimentos futuros nesta área.

Palavras-chave: Avaliação de Desempenho; Frameworks Web; I/O Não Bloqueante;

Contents

List of Figures

Acronyms

DI Dependency Injection 3, 4

IoC Inversion of Control 3

1 Introduction

Nowadays the most web applications need high performance handling a high number users simultaneous with a low/acceptable time for the user process a request. One approach to archive this performance is the scalability. It is defined as ability to increase the number of requests that can be handled simultaneously, when new resources are added [7]. There are two types of scaling a web application:

- scale vertically - add resource to a single server
- scale horizontally - add new servers

In the case of vertical scaling is the more direct approach, but the application need to be design to fully utilize the resources [3]. The horizontal scaling is more expensive because the synchronization overhead [6] but can give benefits like for example fault tolerance.

1.1 Approach and Objectives

In this project, we are constructing a configurable testbed within the Java ecosystem, that allows the specification of various configurations. These configurations include the desired number of nodes for load balancing, the number of worker threads tailored to testing requirements, and other parameters. The architectural representation, as depicted in the Figure 1.1, illustrates the system when viewed from above.

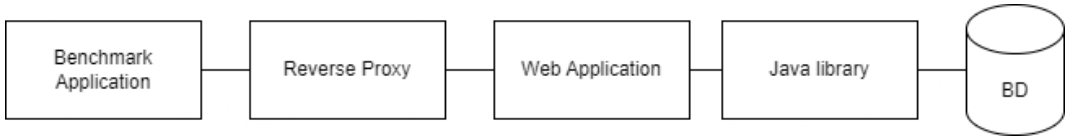


Figure 1.1: *Initial conceptual diagram of the solution*

The primary objective of the project is to test the multi-threaded approach (e.g., Spring MVC) against the event-based approach (e.g., WebFlux). As a secondary objective, the

testing will extend to evaluating other Java event-driven frameworks (e.g., VertX, Akka, etc.). In the existing Java library, there are no implementations of non-blocking I/O. To address this, scaling will be achieved through user-level threads utilizing a thread-pool for I/O, aiming to recreate a library with an asynchronous API. Another secondary objective involves implementing a version that employs a database management driver with non-blocking I/O.

The testbed facilitates experimentation by running multiple instances of the web application across various nodes, allowing observation of how the Reverse Proxy distributes requests. Additionally, this environment is designed to integrate JMeter or JHM as a load testing tool, enabling the measurement of various aspects of application performance.

The platform developed for this dissertation is publicly available on a repository in the GitHub[1]. This repository includes all the source code and scripts necessary for executing the tests, as well as an explanatory guide that instructs users on how to run the performance tests.

1.2 Document organization

This document is structured as follows.

Background and Related Work (Chapter 2) describes research background and provides introduction into main concepts that are being investigated in the following chapters and emphasizes on the summary of related works regarding web application performance.

Solution proposed (Chapter 3) defines the solution goals and research questions. After that some details of the implementation are discussed.

Results (Chapter 4) presents the outcomes of the performance tests conducted using the proposed solution. It includes a comparative analysis between the multi-threaded approach of Spring MVC and the event-driven approach of WebFlux, highlighting key metrics such as response times and throughput under varying load conditions.

Conclusion (Chapter 5) summarizes the findings of the research and discusses their implications for the development of high-performance web applications. It also outlines potential avenues for future research, emphasizing the importance of ongoing exploration in the field of web application performance and scalability.

[1] https://github.com/isel-sw-projects/2023-web-arch-bench

2 Background and Related Work

This chapter gives a short introduction and concepts of some web applications frameworks and benchmark framework used in the work.

2.1 Blocking and non-blocking web frameworks

A request can be handled in two ways: blocking or non-blocking I/O operation [3]. In case of blocking approach when a thread performs a I/O operation, the resources are blocked until the operation is finished. The processor switches to other threads while awaiting for the blocked thread being unblocked. In case of non-blocking it uses something like a event notification to register the request, so the thread can work on something else, instead of waiting the operation completed, because when concluded, an event occurs with the result of the process of the request. For this reason the non-blocking frameworks might be able to handle more requests using fewer threads than with blocking web frameworks [4].

2.2 Framework Spring

The Spring Framework originated in the 2003 as a response to the complexities of Java Enterprise Edition (JEE) development. The framework's core concepts are:

- Inversion of Control (IoC)
- Dependency Injection (DI)

IoC is a programming principle where the control flow of a software application is shifted from the application itself to an external framework or container. In the context of the Spring Framework, IoC involves delegating the responsibility of managing object creation, lifecycles, and dependencies to the Spring IoC container.

DI is a design pattern where the responsibility for providing the required dependencies of a component is shifted from the component itself to an external entity, typically a Spring container. In DI, dependencies are injected into the dependent object, promoting loose coupling, modularity, and easier testing in software applications.

Spring MVC (Model-View-Controller) is a web framework within the Spring ecosystem, adopting the Model-View-Controller architectural pattern for building web applications. It features components like the DispatcherServlet, Controllers, and Views.

Spring WebFlux is a reactive programming framework in the Spring ecosystem, offering asynchronous and non-blocking processing for building web applications. It embraces a reactive paradigm, providing both annotation-based and functional programming models. It features components like WebClient, Flux and Mono classes, and RouterFunctions. Supporting both reactive and traditional Servlet-based APIs.

2.3 Testing Tools: JHM vs Jmeter

In software performance testing, selecting the appropriate tool is essential for accurate results. Two widely used tools in the Java ecosystem are Java Microbenchmark Harness (JMH) and Apache JMeter, each serving different purposes and offering distinct features.

Apache JMeter is an open-source benchmarking tool renowned for its ability to conduct performance testing across web applications, databases, and various protocols (HTTP, HTTPS, FTP, JDBC, etc.). With a user-friendly graphical interface, JMeter supports distributed testing, allowing users to simulate diverse scenarios and assess application performance under varying loads. Its extensive feature set includes measuring response times, throughput analysis, and bottleneck identification, making it ideal for evaluating the overall performance, scalability, and reliability of systems under stress.

Java Microbenchmark Harness (JMH) is a specialized benchmarking tool developed by the OpenJDK community for conducting precise microbenchmarks on Java code. It is specifically designed to measure the performance of small, isolated units of code at a micro-level, such as method latency or algorithm throughput. JMH's integration with the JVM and its warm-up phase contribute to accurate performance measurements by mitigating common pitfalls like JVM warm-up times, JIT compilation effects, and garbage collection. JMH supports various benchmarking modes and allows developers to tailor benchmarks to specific performance aspects, offering precision and control for fine-tuning Java code.

While both JMH and JMeter are valuable tools for performance testing, their use cases are distinct and often complementary. JMH is ideal for developers focused on optimizing specific pieces of Java code, providing a controlled environment to understand the fine-grained performance characteristics at a very detailed level. In contrast, JMeter is suited

for testing the overall performance and scalability of applications by simulating real-world usage patterns and stress conditions, making it a preferred choice for performance engineers who need to understand how an entire system behaves under load.

In practice, JMH is commonly used during the development and optimization phases, where understanding the performance of individual components is crucial. Once the application is ready for deployment, JMeter becomes the tool of choice to ensure that the entire system can handle expected user loads and respond efficiently under various conditions.

Understanding the strengths and limitations of these tools is essential for selecting the right tool for the specific performance testing needs of a project. While JMH and JMeter may appear to serve similar purposes, their differences highlight the importance of choosing the right tool for the job at hand, depending on whether the focus is on micro-level code performance or macro-level system performance.

2.4 Benchmark framework

Benchmark frameworks are essential tools for assessing and comparing the performance of software systems, enabling systematic measurement of metrics like response time and throughput. For this dissertation to simulate a realistic scenario, will be use TPC-C.

TPC-C is a standardized benchmark that simulates a realistic e-commerce application scenario, used for evaluating the performance and scalability of database systems in online transaction processing (OLTP) workloads. The benchmark utilizes a metric called "Transactions per Minute"(TPM) to quantify the number of complete transactions a system can process in one minute.

In the TPC-C benchmark, warehouses are crucial as they simulate distinct business locations, each with its own set of tables for managing stock, orders, and customer data. This setup models the distribution of data across multiple sites, reflecting real-world business operations and allowing for a comprehensive assessment of database performance, including transaction throughput and concurrency control. Each warehouse typically manages data for 10,000 customers and 100,000 items, creating a substantial transactional load. This numerical detail is essential for testing how well a database system handles and queries large volumes of interconnected data. The benchmark's architecture diagram (Figure 2.1) illustrates these relationships and the interactions between warehouses and other tables, highlighting the importance of warehouses in the TPC-C framework.

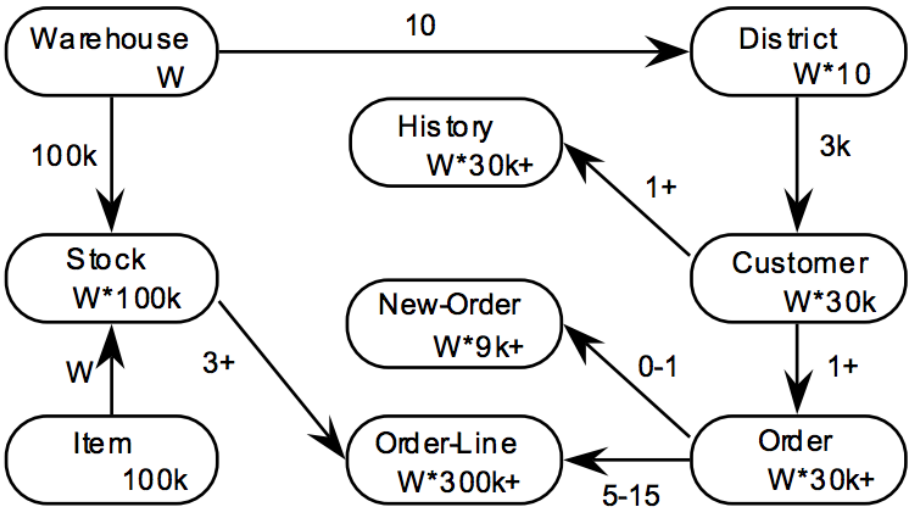


Figure 2.1: *TPC-C workload diagram*[1]

At its core, the TPC-C benchmark models the operations of a wholesale supplier, encompassing multiple aspects of the business, including order processing, payment, order status, stock level, and delivery. Here's a breakdown of how each type of transaction fits into the business context:

New-Order Transactions: This transaction models the process of entering a new order from a customer. It involves updating the inventory, creating an order record, and generating an order status. In a real-world scenario, this mimics the act of a customer placing an order either online or through a sales representative. The system must handle high volumes of such transactions efficiently to ensure customer satisfaction and operational efficiency.

Payment Transactions: This simulates the processing of payments from customers, which includes updating the customer's balance and recording the transaction. Accurate and timely processing of payments is crucial for maintaining cash flow and managing accounts receivable. This transaction reflects the financial interactions between the supplier and its customers.

Order-Status Transactions: This involves checking the status of an existing order. It requires querying the database and retrieving relevant information. Customers and internal staff often need to check the status of orders to track shipments and ensure timely delivery. Efficient handling of these queries is vital for customer service and operational transparency.

Delivery Transactions: This transaction updates the database to reflect the delivery of orders, including adjustments to inventory and customer balances. It represents the

[1] https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf Clause 1.2

final step in the order fulfillment process. Ensuring that this transaction is processed accurately helps maintain inventory accuracy and customer satisfaction by confirming that orders have been delivered.

Stock-Level Transactions: This transaction checks the stock levels for a list of items to ensure inventory availability. Managing inventory levels is critical for avoiding stockouts and overstock situations. This transaction helps businesses maintain an optimal inventory balance, supporting efficient supply chain management.

2.5 Related Work

In this section, we review and analyze two significant theses in our field, providing a comprehensive comparison and highlighting how our research builds on and diverges from these works.

Bilski in his research [1] addresses the challenges of performance and scalability in web applications, particularly those based on Resource Oriented Architecture (ROA) using REST web services. This research was motivated by a real-world problem faced by a software company contemplating a shift to a non-blocking web framework. The objective was to assess the impact of web framework type on performance and to develop guidelines for transitioning from blocking to non-blocking JVM web frameworks. The methodology involved selecting popular web frameworks, conducting performance experiments, consulting with software architects, and formulating migration guidelines. In the testing the benchmark tool run a defined number of simultaneous connections for each web application and for each number the test was repetead 5 times and lasted 30 seconds. The values used to compare the results are the requests per second for each test. Results indicated that non-blocking frameworks could improve performance by up to 2.5 times compared to blocking frameworks, with an average improvement of 27% in a real application. The study concluded that migrating to non-blocking frameworks significantly enhances performance, though further research is needed for broader applicability. The scripts used are no longer public, so the results cannot be replicated.

Catrina's scholarly work [2] examines the demand for highly responsive and scalable web applications due to the rapid increase in internet users and connected devices. Inspired by the author's experience as a developer at Crosskey and their use of Spring WebFlux, this dissertation investigates the performance and scalability of Spring WebFlux compared to traditional synchronous web development. The research aimed to answer key questions regarding Spring WebFlux's performance, scalability, and implementation considerations. The methodology included a detailed review of reactive programming fundamentals, an in-depth exploration of Spring WebFlux's architecture and features, the design and implementation of a non-blocking REST API using Spring WebFlux, and performance evaluation benchmarks against a comparable synchronous API. In this test the values

compared are the mean response time, percentace of succed request, threads of service used, CPU and Java Heap size. Findings revealed that Spring WebFlux significantly enhances concurrency and scalability in web applications, demonstrating the benefits of a non-blocking reactive approach over traditional synchronous methods. This test are explain in the work and is not configurable.

While the aforementioned studies provide valuable insights, they have certain limitations. For example, neither work specifies the benchmark used in their tests, and both offer little to no configurability. In contrast, our approach provides a configurable benchmark that can be extensively applied to other frameworks by implementing a web application contract. Additionally, unlike Catrina's work, where the API created for the tests does not represent real-world scenarios, our work will be based on TPC-C, a standard On-Line Transaction Processing Benchmark that simulates a real case of an online shop. This ensures that the tests will primarily focus on I/O operations. Furthermore, our work will test the use of a reverse proxy to compare a blocking framework with a non-blocking framework in a distributed environment.

3 Proposed solution

In this chapter, we present the comprehensive solution developed to serve as a testbed for benchmarking multiple APIs, using the TPC-C benchmark as a representative scenario. The solution is implemented through a detailed architecture designed to integrate various components, ensuring robust performance and scalability. This architecture is organized into distinct areas: the Benchmark Application, the Web Application, and the TPC-C Java library. The architecture allows for flexible configuration. This flexibility enables the testbed to accommodate different APIs, databases and Benchmark configurations, facilitating a broad evaluation and comparison under standardized conditions.

3.1 Solution Architecture

The solution proposed is a testbed in JVM environment. For the first minimum viable product (MVP) are implemented what is in Figure 3.1.

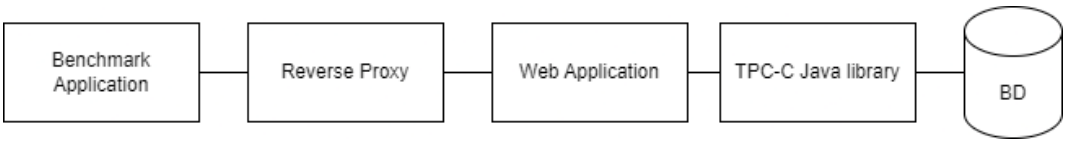


Figure 3.1: *Block diagram illustrating the conceptual structure of the solution*

3.1.1 Benchmark Application

JMeter utilizes the concept of concurrent requests as Active Threads. To differentiate between this concept and the threads within the Web Application, we will refer to them as JMeter Threads.

The tests begin with a ramp-up period for each iteration and run for the duration defined. For eacg iterations the number of simultaneous requests changed by the values defined in the configuration.

3.1.2 Web Application

The web application is the object of study that will be tested.

To conduct these tests, it is necessary to implement specific routes corresponding to operations from the TPC-C Java library.

The routes to be implemented are as follows:

- POST /api/stock: This route will handle stock-related operations, such as updating inventory levels.
- POST /api/payment: This route will process payment transactions, including recording payments and updating customer balances.
- POST /api/order: This route will manage the creation and processing of orders.
- GET /api/order/status: This route will retrieve the status of a specific order, providing details such as fulfillment or shipment status.
- POST /api/delivery: This route will manage the delivery process, including updating delivery status and tracking information.

By implementing these routes, the web application will be ready for testing against the TPC-C benchmarks, ensuring that all required operations are supported.

3.1.3 TPC-C Java library

For this library, a specific repository from GitHub has been used as the base. [1] This repository provides operations using JDBC, which means it only supports synchronous operations.

To implement asynchronous operations, we are utilizing Kotlin's coroutines. Kotlin coroutines are lightweight threads that enable concurrent execution of multiple tasks within a single thread. By leveraging a dispatcher, particularly the IO dispatcher with a specialized thread pool, coroutines manage the execution of tasks in a non-blocking manner. This allows the web server to perform a context switch to the IO dispatcher, launching a new coroutine while freeing up the web server thread to handle other HTTP requests. This approach ensures compatibility with both synchronous and asynchronous APIs. Moving forward, replacing JDBC with an asynchronous driver would enable true asynchronous operations and eliminate the limitations of synchronous database operations.

The database creation is based on a more recent repository from GitHub. [2]. To comply with the TPC-C framework, the number of warehouses is specified in the configuration.

[1] https://github.com/AgilData/tpcc
[2] https://github.com/Li-Xiang/tpcc-jdbc

The code then generates random data to populate the other tables in the schema. The database used is H2 in-memory, but it can be replaced with another database by passing the appropriate connection string.

3.2 Customization and configuration

The flexibility of the testbed is enhanced through a JSON configuration file, which encompasses all settings required to tailor the testbed to specific needs. This configuration file allows for adjustments to parameters such as database connections, API ports, NGINX paths and configurations of the benchmark, enabling users to adapt the testbed for various scenarios.

You can see an example of that JSON configuration file below:

```
{
    "db": {
      "port": "9090",
      "jdbcUrl": "jdbc:h2:mem:mydb;MODE=MYSQL",
      "username": "sa",
      "password": "",
      "database_driver" : "H2",
      "warehouse_number": 1,
      "driver_class_name": "org.h2.Driver",
      "max_connections_pool": 100,
      "max_connections_idle": 10,
      "min_connections_idle": 5,
      "max_wait_millis_conn": 10000
    },
    "api": {
      "ports": [8081,8082,8083,8084],
      "project": "spring-webflux"
    },
    "benchmark":{
      "result_path": "target//Results//WebFluxNginx4",
      "simultaneous_requests": [10,100,500,1000,2000],
      "durationSeconds":50,
      "rampUpSeconds": 10
    },
    "ngnix": {
      "enable": true,
      "port": 8080,
      "path": "../../../nginx-1.25.4",
      "conf": "../../../nginx-1.25.4/conf"
    },
    "url": "jdbc:h2:tcp://localhost:9090/mem:mydb",
    "asyncParallelismDispatcher": 128
}
```

Additionally, a pre-defined template is provided for NGINX, which can also be adjusted to fit web server settings. This template includes default settings for optimizing the server to handle different workloads and API interactions. Users can modify this template to better suit their particular environment and requirements. It is complemented by the ports specified in the JSON configuration file and starts NGINX on port configurable.

You can find the template for the NGINX file below:

```
worker_processes auto;  # Based on available CPU cores
worker_rlimit_nofile 4096;
worker_rlimit_core unlimited; #Comment when running in Windows
http{
    include mime.types;
    keepalive_timeout 100;
    client_header_timeout 12;
    send_timeout 60;
    client_body_buffer_size 16K;
    client_max_body_size 10M;
    client_body_timeout 12;
    proxy_buffer_size 16k;
    proxy_buffers 8 16k;
    proxy_busy_buffers_size 16k;
    upstream spring {
         {{upstreamBlock}}
         keepalive 32;
    }
    server {
        listen {{nginxPort}};
        location / {
            proxy_pass http://spring/;
            proxy_connect_timeout 120s;
            proxy_send_timeout 120s;
            proxy_read_timeout 120s;
            send_timeout 120s;
        }
    }
}
events {
    worker_connections 4096;
}
```

To switch to a different web API, users must implement an API that adheres to the specifications provided by the TPC-C Java Library. This requirement ensures that the new API is compatible with the benchmark's protocols and can be seamlessly integrated into the testbed. By following these guidelines, users can efficiently evaluate different APIs and databases while maintaining consistent benchmarking standards.

There are two scripts: one in batch for Windows users and another in shell for Mac and

Linux users. Both scripts are located in the folder `scripts`. [3]

The scripts automates the setup and configuration of the testbed using settings from the `benchmark_setting.json` file. It retrieves configuration details such as API ports, project directories, and NGINX paths, and then starts the database server and API services on the specified ports. The script performs health checks on the APIs and, if necessary, retries until all services are operational. It also manages NGINX by stopping any existing instance, updating the configuration with new settings, and restarting the server. Finally, the script initiates the benchmarking process, ensuring that all components are correctly configured and ready for testing.

[3] https://github.com/isel-sw-projects/2023-web-arch-bench/tree/main/web-arch-bench/scripts

4 Results

In this chapter, we will analyze the results. Using the same workload, we compare Spring MVC and WebFlux, while modifying the APIs and some NGINX configurations. We will compare the number of transactions per second, varying by the number of simultaneous requests. All test configurations mentioned in this section are available on GitHub in the `config/examples` folder. [1]

The machine used to run this test is a PC with Windows 11, an Intel® Core™ i5-10400F processor, and 48GB of RAM.

4.1 Workload Configuration

The workload for the tests is defined in the field `benchmark` in the configuration JSON with the following values:

```
"benchmark": {
    "result_path": "target//Results",
    "simultaneous_requests": [10, 100, 500, 1000, 2000],
    "durationSeconds": 50,
    "rampUpSeconds": 10
},
```

The parameter `simultaneous_requests` is an array that defines the number of simultaneous JMeter threads. The `durationSeconds` parameter specifies the duration for each set of simultaneous requests, while `rampUpSeconds` defines the time taken to reach the specified number of simultaneous requests.

[1] https://github.com/isel-sw-projects/2023-web-arch-bench/tree/main/web-arch-bench/config/examples

4.2 Spring MVC

In this section, we will discuss the tests for Spring MVC. We will compare the results for cases with no reverse proxy, as well as with a reverse proxy using 2 and 4 instances of the MVC API. Due to some limitations between Windows and NGINX, we repeated the tests with NGINX running on Ubuntu in the Windows Subsystem for Linux (WSL).

WSL (Windows Subsystem for Linux) is a compatibility layer in Windows that allows users to run a Linux distribution natively, without the need for a virtual machine or dual boot. While WSL provides near-native performance, it introduces some overhead due to the abstraction layer between Windows and Linux. This can impact aspects such as I/O operations, network performance, or system calls, which may explain the lower transaction rate compared to the baseline. To better understand the impact of WSL, future tests could be conducted on a native Linux environment, where such overhead would not be a factor.

For this test, we are using the default thread pool size, which is 200 threads in the case of Spring MVC.

4.2.1 No reverse proxy

This test serves as a baseline to assess the performance of Spring MVC without load balancing or proxying. The stable transaction rate demonstrates the application's performance under direct requests. In this test, only one instance is running on port 8081, and NGINX is not enabled. The results are shown in the following image:

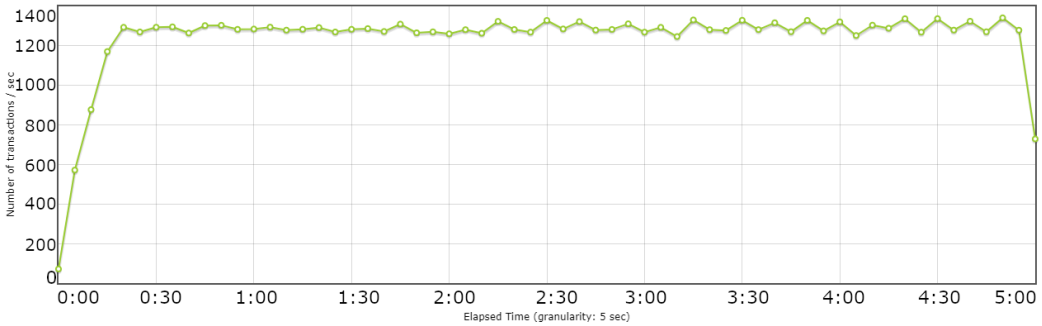


Figure 4.1: *Spring MVC without Reverse Proxy (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

In this test, no transactions failed, and the number of transactions per second remained steady at around 1300 requests. This consistent transaction rate suggests that the application can handle the load efficiently without any major performance degradation. The result could be influenced by factors such as the underlying hardware, JVM tuning, or the absence of network bottlenecks.

In the following subsections, will compare these results with those obtained when using NGINX as a reverse proxy, to evaluate the impact of load distribution on performance.

4.2.2 Reverse Proxy 2 APIs

Due to limitations on Windows, this test was also run once on Ubuntu in WSL. The test is executed on Windows, with two instances running on ports 8081 and 8082, and NGINX enabled and running on port 8080. The results are shown in the following image

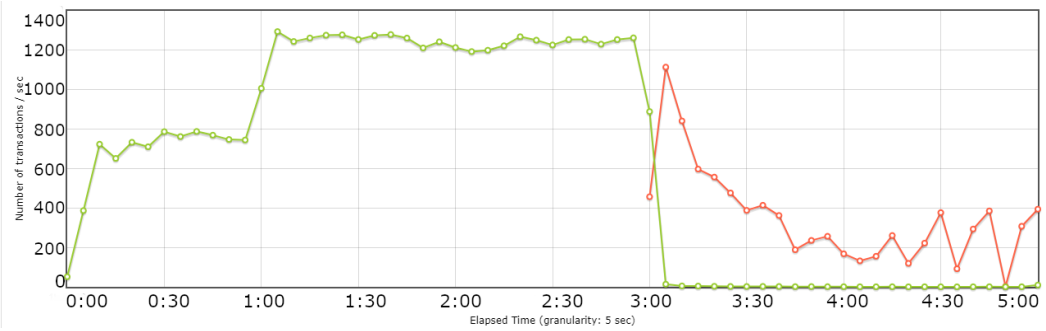


Figure 4.2: *Spring MVC with NGINX enabled and 2 API instances in Windows (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

In this graph, we can see that during the first minute (10 simultaneous requests), the number of transactions is nearly 800 requests per second. In the second and third minutes (100 and 500 simultaneous requests), the rate increases to between 1200 and 1300 requests per second. However, in the fourth and fifth minutes (1000 and 2000 simultaneous requests), NGINX crashes, giving an error that cannot be recovered. This error is likely due to NGINX running in single-worker process mode on Windows, which limits the server's ability to handle high concurrency and fully utilize multiple CPU cores, thereby affecting performance under heavy load.

In the next graph, we repeat the same test, but on Ubuntu in WSL. In this Linux environment, NGINX can use multiple worker processes and leverage event-driven mechanisms like epoll or kqueue to handle a large number of connections.

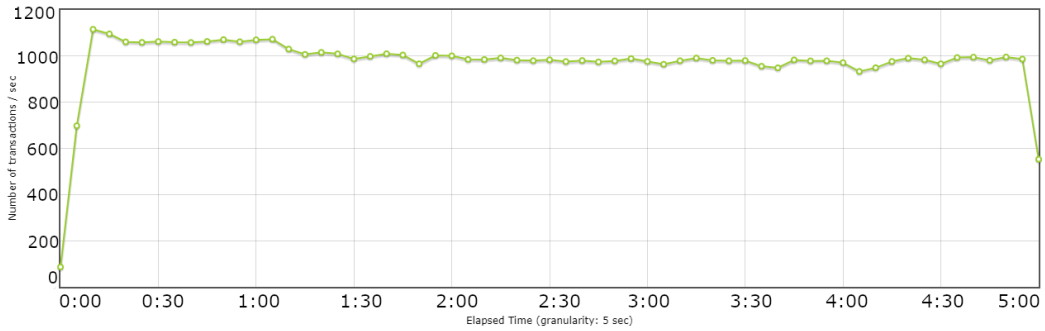


Figure 4.3: *Spring MVC with NGINX enabled and 2 API instances in WSL (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

In this test, no transactions failed, and the number of transactions per second remained steady between 1000 and 1100 requests. This is lower than our baseline, likely due to the additional variable introduced by WSL. Although the performance remains stable, the reduced transaction rate suggests some overhead that affects overall efficiency.

4.2.3 Reverse Proxy 4 APIs

The first test is executed on Windows, with four instances running on ports 8081 to 8084, and NGINX enabled and running on port 8080. The results are shown in the following image

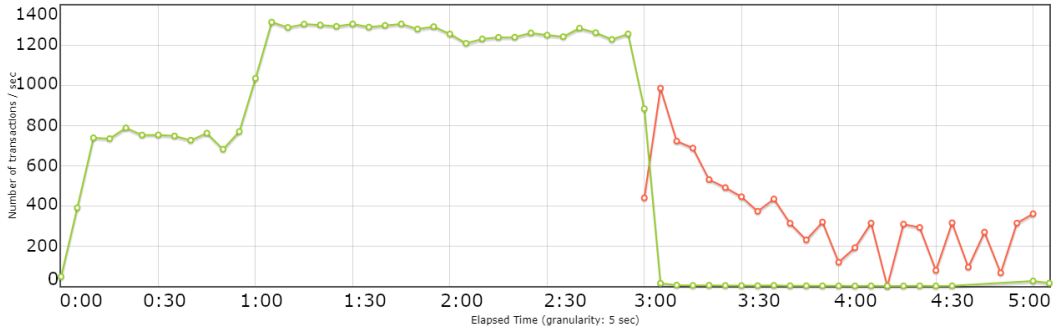


Figure 4.4: *Spring MVC with NGINX enabled and 4 API instances in Windows (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

This graph is very similar with the Figure **??** from the previous section, leading to a consistent conclusion: NGINX efficiently distributes requests across the API instances up to 1000 simultaneous requests and beyond, while maintaining stable performance regardless of the number of instances. However, further analysis could reveal whether increasing the number of instances beyond a certain point might introduce diminishing returns or additional overhead.

In the next graph, we repeat the same test, but on Ubuntu in WSL.

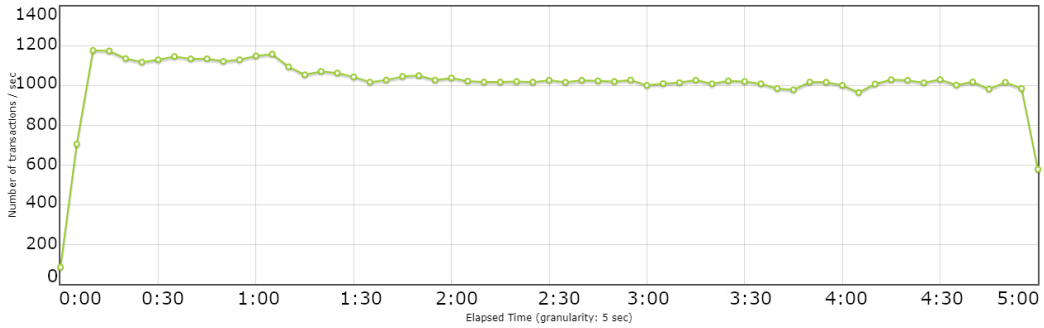


Figure 4.5: *Spring MVC with NGINX enabled and 4 API instances in WSL (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

The performance in this test is similar to the results with 2 API instances, but with superior overall performance. This improvement is likely due to the increased capacity provided by the additional API instances, allowing NGINX to distribute the load more effectively. The higher transaction rate indicates that the system can handle more simultaneous requests when scaling to 4 instances, suggesting that the additional instances help mitigate the impact of high concurrency. However, the performance gain should be further analyzed to determine if it scales efficiently beyond this point or if the system begins to experience diminishing returns or bottlenecks.

4.3 Spring Webflux

In this section, similar to the previous one on Spring MVC, we will discuss the tests for Spring WebFlux. We will compare the results for cases without a reverse proxy, as well as with a reverse proxy using 2 and 4 instances of the WebFlux API. Due to limitations in NGINX's ability to handle high concurrency on Windows, we repeated the tests with NGINX running on Ubuntu in the Windows Subsystem for Linux (WSL). By running the tests in WSL, we aim to eliminate the performance bottlenecks introduced by NGINX's limitations on Windows.

For this test, we are using the default thread pool size of 24 threads (calculated as logical processors * 2) in the case of Spring WebFlux.

4.3.1 No reverse proxy

This test serves as a baseline to assess the performance of Spring MVC without load balancing or proxying. The stable transaction rate demonstrates the application's performance under direct requests, but as the workload approaches 1000 simultaneous requests, failures begin to occur, and performance becomes less stable. This instability could be attributed to the use of a dispatcher rather than a fully asynchronous database driver,

which may limit the system's ability to handle high concurrency. In this test, only one instance is running on port 8081, and NGINX is not enabled.

These results will serve as a baseline for comparison with future tests involving load balancing or multiple instances. The results are shown in the following image:"

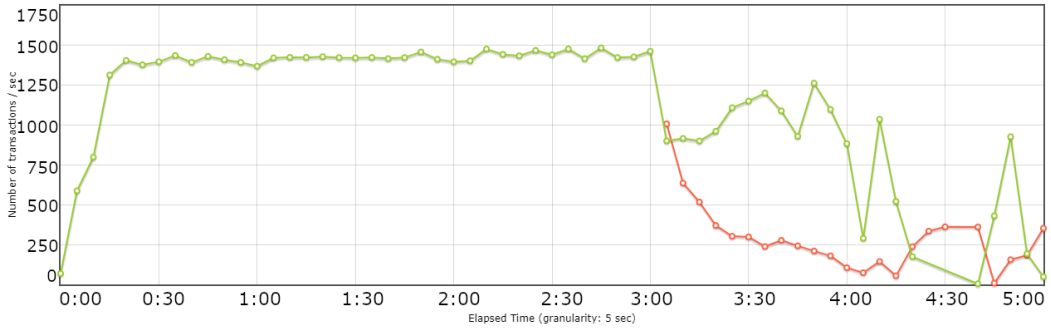


Figure 4.6: *Spring Webflux without Reverse Proxy (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

In this image, the transaction success rate remains stable up to a point, but as the number of simultaneous requests approaches 1000 or more, transaction failures begin to occur. This performance degradation is likely due to a configuration the configuration `asyncParallelismDispatcher` is set to 128. This setting limits the concurrency level, meaning that beyond a certain number of requests, the dispatcher is unable to efficiently manage the workload. This suggests that increasing that config value or using a more fully asynchronous configuration might improve stability under higher loads.

4.3.2 Reverse Proxy 2 APIs

Due to limitations on Windows, this test was also run once on Ubuntu in WSL. The test is executed on Windows, with two instances running on ports 8081 and 8082, and NGINX enabled and running on port 8080. The results are shown in the following image

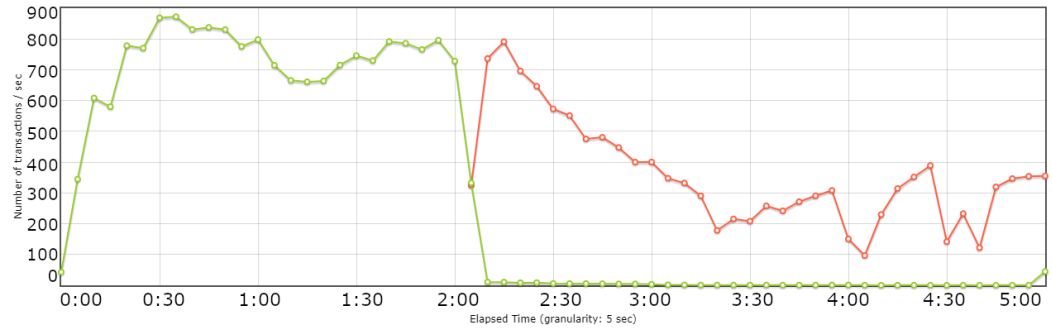


Figure 4.7: *Spring Webflux with NGINX enabled and 2 API instances in Windows (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

In this test, compared to the baseline, there is a drop in performance during the first two minutes (with 10 and 100 simultaneous requests), likely due to the overhead introduced by NGINX as a load balancer. This overhead could be due to the extra processing required by NGINX to distribute requests between multiple instances and manage connections, particularly with smaller workloads. At the start of the third minute (500 simultaneous requests), failures begin to occur earlier than in the baseline test, where failures only started around 1000 simultaneous requests. This earlier failure is probably due to the additional load-balancing overhead.

In the next graph, we repeat the same test, but this time on Ubuntu in WSL. As expected, similar to the results in subsection 4.2.2, NGINX does not fail in this test. This was anticipated because WSL allows NGINX to run in a Linux environment, where it can utilize multiple worker processes and handle concurrency more efficiently than on Windows. The improved performance in WSL suggests that NGINX's limitations on Windows might be a significant factor in the earlier test failures, indicating that running NGINX in a native Linux environment could lead to more stable performance under high load conditions.

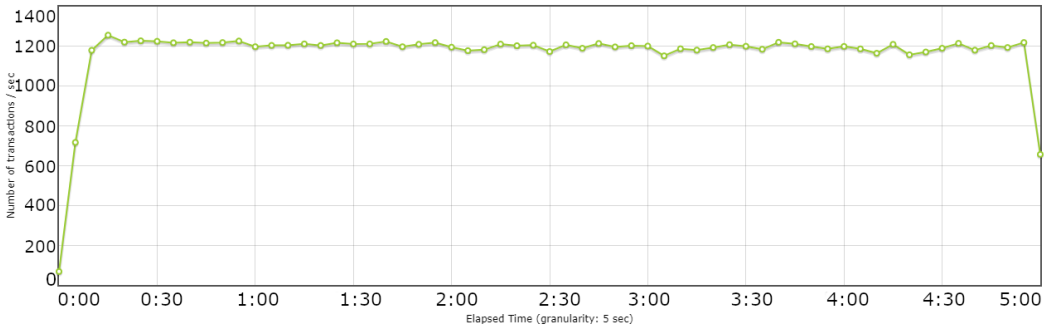


Figure 4.8: *Spring Webflux with NGINX enabled and 4 API instances in WSL (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

4.3.3 Reverse Proxy 4 APIs

The first test is executed on Windows, with four instances running on ports 8081 to 8084, and NGINX enabled and running on port 8080. The results are shown in the following image

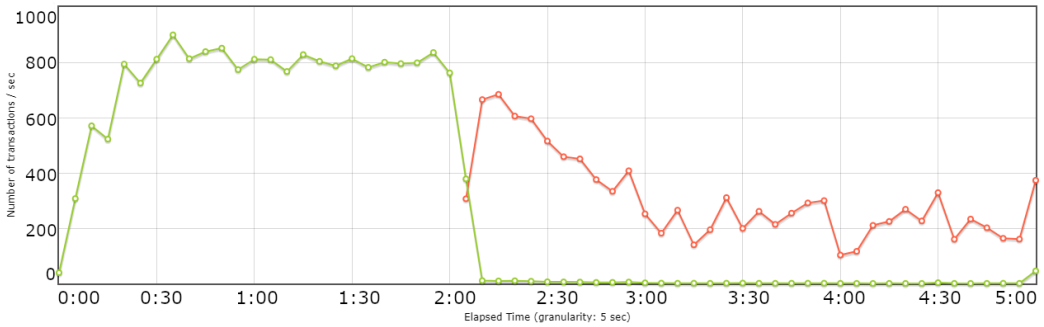


Figure 4.9: *Spring Webflux with NGINX enabled and 4 API instances in Windows (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

This test, compared to the WebFlux test with NGINX enabled and two API instances on Windows, shows a slight improvement in the first two minutes (with 10 and 100 simultaneous requests), reflected in a marginally higher transaction rate during this period. However, the failure still occurs at the same point as in the previous test, around 500 simultaneous requests. Despite the initial improvement, the failure at the same point could be due to underlying limitations in how the system handles high concurrency, such as resource exhaustion or configuration constraints.

In the next graph, we repeat the same test, but this time on Ubuntu in WSL.

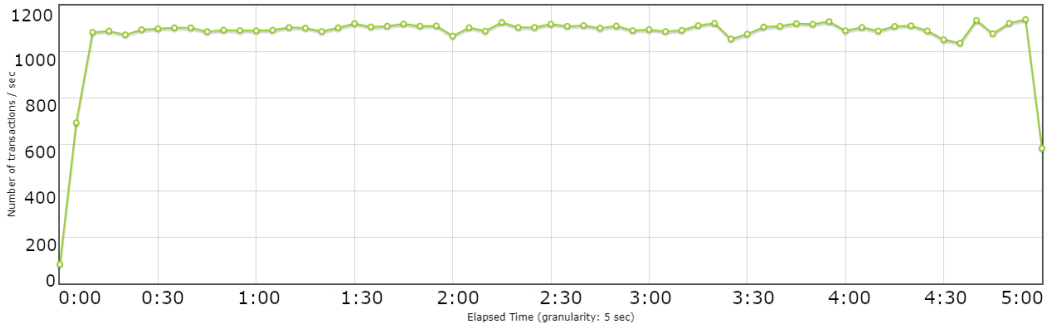


Figure 4.10: *Spring Webflux with NGINX enabled and 4 API instances in WSL (Transactions per second over time). Green represents successful transactions, while red indicates failed transactions.*

As expected, the failure at 500 simultaneous requests was prevented, but the number of transactions per second is slightly lower when using four API instances compared to previous tests. This slight reduction in transactions per second might be due to the overhead introduced by load balancing across the instances, or due to suboptimal resource allocation between instances.

While preventing failure at 500 simultaneous requests is a significant improvement, the lower transaction rate suggests that further optimization of the load-balancing configuration or system resources may be necessary to achieve both stability and high throughput. Compared to the single-instance configuration, this setup provides more stability but at

the cost of slightly reduced performance, highlighting a trade-off between concurrency management and transaction throughput.

5 Conclusion

The results presented throughout this dissertation demonstrate the effective implementation of a flexible and reproducible testbed for benchmarking JVM-based web architectures, focusing not only on Spring MVC and WebFlux but also on the underlying performance characteristics of the JVM under various configurations. This testbed, which leverages JMeter for load generation, along with NGINX for load balancing, offers a comprehensive framework to evaluate both blocking and non-blocking I/O frameworks, particularly under high-concurrency workloads.

By using multiple API instances and running tests across different environments (Windows and WSL), we were able to identify performance bottlenecks, such as NGINX's limitations on Windows when handling high concurrency, and the effects of increasing API instances on transaction throughput. The tests performed in WSL further highlight the benefits of Linux's better handling of concurrent connections compared to Windows.

The testbed developed in this research provides a robust platform for JVM-based performance analysis and opens several avenues for future work, which are discussed in the following section.

5.1 Future Work

This dissertation has laid the groundwork for further research in the area of JVM-based web architecture benchmarking. Below are some key areas for future exploration and improvement:

- **Native Linux Testing**: While WSL offers a Linux-like environment on Windows, testing on a fully native Linux system could yield more accurate performance data by eliminating the overhead introduced by WSL. This would ensure better alignment of the results with real-world deployments where Linux is the preferred environment for high-concurrency web applications.

- **Optimizing NGINX Configurations**: Further optimization of NGINX configurations for handling high loads without performance degradation could improve results. Areas of focus could include fine-tuning worker processes, connection handling mechanisms (e.g., using `epoll`), and experimenting with different load-balancing algorithms.
- **Performance Tuning in JVM-based Applications**: Future research could explore JVM tuning (such as garbage collection and heap size configurations) and application-level optimizations, especially for Spring MVC and WebFlux, to improve throughput and reduce latency under high-concurrency scenarios.
- **Fully Asynchronous Database Driver**: Another promising area of research would be the integration and evaluation of a fully asynchronous database driver. This would allow the system to leverage the full potential of non-blocking I/O, particularly in WebFlux, further improving the scalability and responsiveness of the system under high-concurrency database loads.
- **Testing Other JVM-based Frameworks**: Extending the testbed to benchmark additional JVM-based frameworks, such as Vert.x, would provide valuable insights into how they compare with Spring MVC and WebFlux under various load conditions. Vert.x, with its event-driven, non-blocking model, could offer interesting performance contrasts, particularly in high concurrency scenarios.
- **Java 21 Virtual Threads**: With the introduction of virtual threads in Java 21 as part of Project Loom, future work could explore their impact on the performance of both blocking and non-blocking frameworks. Virtual threads provide lightweight concurrency management and may offer a compelling alternative to traditional thread pools, potentially simplifying the design of scalable applications.
- **Cloud Environments**: Extending the testbed to cloud environments (e.g., AWS, GCP, or Azure) would provide valuable insights into how these JVM-based web architectures perform under cloud-native conditions. This would include auto-scaling, containerization, and other cloud-specific optimizations and limitations.

In conclusion, this dissertation successfully demonstrates the development and validation of a robust and reproducible testbed, which is built to benchmark JVM-based web architectures. The testbed is made publicly available through a repository, enabling other researchers and practitioners to replicate all the tests and experiments conducted. This open access serves as a solid foundation for future research and experimentation. Further optimizations and extensions, as outlined above, will allow for a deeper understanding of the performance characteristics of both blocking and non-blocking frameworks under various configurations.

Bibliography

[1] M. Bilski. "Migration from Blocking to Non-Blocking Web Frameworks". Dissertation. Blekinge Institute of Technology, 2014. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-5932 (cit. on p. 7).

[2] A. Catrina. "A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development". Thesis. HAMK University of Applied Sciences, 2023. URL: https://www.theseus.fi/bitstream/handle/10024/812448/Catrina_Alexandru.pdf?sequence=2 (cit. on p. 7).

[3] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson. "Improving the scalability of a multi-core web server". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013, pp. 161–172 (cit. on pp. 1, 3).

[4] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. "Comparing the performance of web server architectures". In: *ACM SIGOPS Operating Systems Review* 41.3 (2007), pp. 231–243 (cit. on p. 3).

[5] M. Pato. *The ISELthesis LaTeX Template's Manual*. Instituto Superior de Engenharia de Lisboa (ISEL-IPL). 2024. URL: https://github.com/matpato/iselthesis (cit. on p. viii).

[6] S. Tilkov and S. Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83 (cit. on p. 1).

[7] B. Veal and A. Foong. "Performance scalability of a multi-core web server". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. 2007, pp. 57–66 (cit. on p. 1).